

# 1

# Einführung

## ■ 1.1 Aufteilung dieses Buches

Dieses Buch ist, nach einer kurzen Einführung in diesem Kapitel, in die für das Software-Engineering wichtigen **Software-Lebenszyklusphasen** untergliedert. So enthält jedes Kapitel den Titel einer dieser Phasen. Neben einer sinnvollen Strukturierung des Buches bekommt der Leser dadurch auch einen zeitlichen Plan, wie ein Softwareentwicklungsprojekt in der Regel abläuft.

Zum besseren Überblick werden in Kapitel 2 vorab phasenübergreifende Verfahren besprochen und eingeschoben.

In Kapitel 3 wird Software-Engineering in der **Planungsphase** erörtert.

Das sogenannte „Requirements Engineering“ (auf Deutsch: Anforderungsanalyse) wird in Kapitel 4 – der **Definitionsphase** – genauer beleuchtet.

In Kapitel 5 folgt die Besprechung der Verfahren für die **Designphase** des Software-Lebenszyklus. Wie gute Softwarearchitekturen erdacht und gebaut werden können, wird hinterfragt.

Die nachfolgende Phase im Software-Lebenszyklus, die **Implementierung**, ist nicht Gegenstandes dieser Arbeit und kann in einschlägiger Literatur über Programmierung nachgelesen werden. Je nach Programmiersprache eignen sich hier unterschiedliche Bücher. Wer jedoch eine Übersicht über mehrere gängige Programmiersprachen sucht, der findet im „Taschenbuch Programmiersprachen“ ([HV07]) ein gutes Nachschlagewerk. Auch bei der Implementierung gibt es natürlich Themengebiete, die in den Bereich Software-Engineering fallen. Beispiele sind hier Versionsverwaltung, Konfigurationsmanagement oder auch die Methoden, um Softwarearchitekturen in sauberen Programmcode umzusetzen. Gute Literatur dazu kann gefunden werden bei Sommerville [Som18] oder beispielsweise zum Thema „Clean Code“ bei Robert C. Martin [Mar08]

Kapitel 6 widmet sich der **Test- und Abnahmephase** und damit den Qualitätssicherungsfragen eines Softwareprojektes.

Zuletzt wird in Kapitel 7 noch die **Wartung**, und damit die wichtigsten Wartungstechniken und Verfahren aus dem Software-Engineering-Bereich, besprochen.

## ■ 1.2 Überblick und Terminologie

### Was ist Software-Engineering?

Dieses Buch widmet sich bekanntlich dem Thema **Software-Engineering**. Dabei handelt es sich um die Sammlung und Beschreibung von Erfolg versprechenden Verfahren, Methoden und Werkzeugen rund um den Prozess von qualitativ hochwertiger Software-Erstellung.



#### Definition

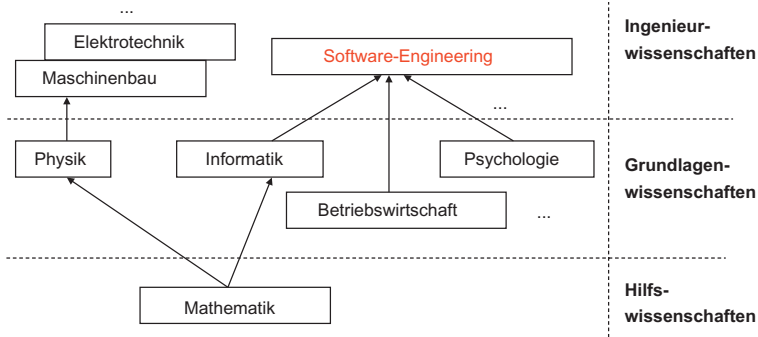
„The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.“

- *IEEE Standard Glossary of Software-Engineering* -[IEEE18]

Gegenstand des Software-Engineering ist die **ingenieurmäßige** Entwicklung **komplexer Softwaresysteme** hoher **Qualität** unter Berücksichtigung der einzusetzenden Arbeits- und Zeitrressourcen.

### Gebietseinordnung

Nach Broy gehört Software-Engineering zu den Ingenieurwissenschaften (siehe Abbildung 1.1) wie beispielsweise auch Elektrotechnik oder Maschinenbau. Diese Ingenieurwissenschaft bedient sich unter anderen der Grundlagenwissenschaften der Informatik, der Betriebswirtschaft und der Psychologie. Die Informatik liefert dabei alle technologischen Grundlagen für Softwaresysteme. Betriebswirtschaftliche Grundlagen sind notwendig, da Projekte betriebswirtschaftlichen Regeln (z. B. ein finanzieller Rahmen) unterliegen und Projekte meist auch in Unternehmen initiiert werden. Grundlagen aus der Psychologie sind erforderlich, um Teamarbeit im Rahmen von Software-Engineering überhaupt erst zu ermöglichen. Ein erfolgreicher Projektverlauf ist erst durch den Erwerb der sogenannten „Soft Skills“ (gemeint sind Fähigkeiten bezüglich Problemlösungen, Konfliktmanagement, Führungsmanagement, etc.) möglich.



**Abbildung 1.1** Gebietseinordnung nach Broy und Rombach [BR02, pp. 438–451]



Die hier verwendete Gebietseinordnung nach Broy wird in Fachkreisen dennoch kontrovers diskutiert, denn Viele sehen die Informatik selbst auch als Ingenieurwissenschaft an und Software-Engineering als Teil der Informatik. Auch verwendet ein Großteil der deutschsprachigen Fachwelt im Deutschen die Übersetzung „Software-technik“ für Software-Engineering (z. B. Informatik-Handbuch [ZR06]).

## Terminologie

In dieser Arbeit werden die nun genannten Begrifflichkeiten und Artefakte wie in Tabelle 1.1 verwendet.

**Tabelle 1.1** Terminologie dieses Werkes

Begriff	Erklärung	Abbildung
Software	„Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.“ IEEE Standard Glossary of Software-Engineering [IEE18]	
Software-System	Ein System, dessen Subsysteme und Komponenten aus Software bestehen, ist ein Software-System.	
Software-Produkt	Ein Produkt ist ein in sich abgeschlossenes, i. A. für einen Auftraggeber bestimmtes Ergebnis eines erfolgreich durchgeführten Projekts (oder Herstellungsprozesses). Ein Software-Produkt ist ein Produkt, das aus Software besteht.	

Begriff	Erklärung	Abbildung
CASE	<p>Dies ist eine Abkürzung (engl.) für „Computer Aided Software-Engineering“ und bezeichnet einen Fachbegriff aus dem Software-Engineering. Gemeint sind hier meist die Werkzeuge, welche eingesetzt werden können, um Software-Engineering zu unterstützen. Beispiele sind UML-Tools wie Visual Paradigm (von Visual Paradigm International [Int19]), Rational Rhapsody Developer (von IBM [IBM19b]) oder Enterprise Architect (von Sparx [Eur19]). Inhaltlich steht die Modellierung von Projektlösungen im Mittelpunkt. Beispiele hierfür sind:</p> <ul style="list-style-type: none"> <li>▪ UML</li> <li>▪ Transformation in verschiedene Programmiersprachen</li> <li>▪ Datenstrukturen</li> <li>▪ Reverse Engineering</li> <li>▪ Generierung von Prototypen.</li> </ul> <p>Aber auch Projektmanagement, Prozessmanagement und viele weitere Entwicklungswerkzeuge sind oft zusätzlich mit im Paket.</p>	
UML [OMG19b]	<p>Es handelt sich um eine Abkürzung (engl.) für „<b>Unified Modeling Language</b>“. Die UML ist eine Modellierungssprache. „Die UML dient zur Modellierung, Dokumentation, Spezifizierung und Visualisierung komplexer Systeme, unabhängig von deren Fach- und Spezialisierungsgebiet. Sie liefert die Notationselemente gleichermaßen für die statischen und dynamischen Modelle von Analyse, Design und Architektur und unterstützt insbesondere objektorientierte Vorgehensweisen. ... Die 1989 gegründete OMG (Objekt Management Group) – ein Gremium mit heute 800 Mitgliedern – ist Hüterin dieses Standards. Das es sich bei Werken der OMG um herstellerneutrale Industriestandards handelt, gewährleistet die Teilnahme aller relevanten Marktvertreter (zum Beispiel Rational Software [IBM], Hewlett-Packard, Daimler AG, I-Logix, Telelogic, Oracle, Microsoft, ...).“ eine breite Unterstützung in der Industrie. (aus [RQSG12, S. 4])</p>	

## Hinweis

Da dieses Buch eine kompakte Übersicht über Software-Engineering bietet und absichtlich viele Themen daher lediglich anschnidet, aber nicht vollständig diskutiert, wird hier und im Verlauf des Buches auf passende ausführlichere Literatur verwiesen. Ein Nachschlagewerk der ausführlichen Natur ist das Buch von Sommerville [Som18] und eine etwas unbürokratische, amerikanische Variante das Buch von Pressman [Pre14]. Zum guten Studium der Modellierungssprache UML sind die „UML Kurzreferenz“ [ÖS14] und „UML glasklar“ [RQSG12] hilfreich.

## 1.2.1 Der Software-Lebenszyklus

Im Fokus von Software-Engineering steht der sogenannte **Software-Lebenszyklus**. Gemeint ist damit der gesamte Prozess, der zur Erstellung und Erhaltung eines Softwaresystems führt. Bei jeder Art von Softwareerstellung läuft dieser Lebenszyklus ab. Er ist zur einfacheren Unterscheidung der einzelnen Tätigkeiten in sogenannte Phasen unterteilt (siehe Abbildung 1.2).

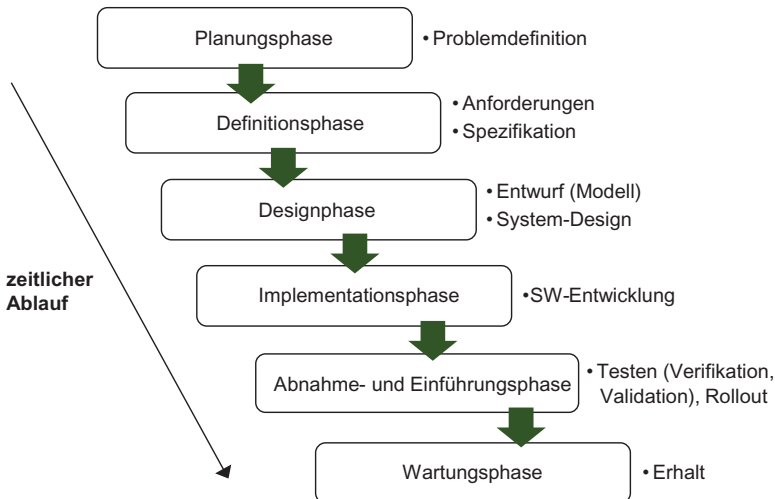


Abbildung 1.2 Software-Lebenszyklus

- **Planungsphase** (s. Kap. 3):  
In dieser Phase wird ein Softwareprojekt neu aufgesetzt, Projektmanagement begonnen, das Lastenheft geschrieben und eine Problemdefinition erarbeitet.
- **Definitionsphase** (s. Kap. 4):  
Nun werden die Anforderungen an eine Software eruiert und in einer Spezifikation dokumentiert.
- **Designphase** (s. Kap. 5):  
Danach erfolgt der Entwurf und die Modellbildung für das Systemdesign der Software. Es entsteht die Softwarearchitektur.
- **Implementationsphase** (vgl. [HV07]):  
In dieser Phase findet die eigentliche Programmierung der Software statt. Zugrunde liegt die in der vorherigen Phase gefundene Softwarearchitektur.
- **Abnahme-/Einführungsphase** (s. Kap.6):  
Um hohe Qualität zu gewährleisten, finden nun das Testen, die Verifikation, die Validation und die Markteinführung (engl. **Rollout**) der Software statt.
- **Wartungsphase** (s. Kap. 7):  
Die Software wird in dieser Phase vor Software-Alterung (engl. Aging) bewahrt; in der Regel werden Fehler der Software gesammelt und wenn möglich behoben. Auch Änderungsanforderungen werden erhoben, um für die aktuelle oder die Folgeversion der Software Verbesserungen erzielen zu können.

Nach der Wartungsphase findet in der Regel entweder die Entwicklung von **Folgeversionen** der Software, manchmal auch deren **Neuentwicklung** oder die **Stilllegung** statt. Bei einer Stilllegung wird fallweise ein sogenanntes **End-of-Life-Management** durchgeführt.

## 1.2.2 Komplexität der Softwareentwicklung

Benutzer von Software erwarten heute einwandfrei funktionierende, effiziente Programme. Software, die nicht den Erwartungen entspricht, wird daher schnell verworfen und Konkurrenzprodukte werden eingesetzt. Softwareproduzenten haben daher großes Interesse daran, qualitativ hochwertige Programme herzustellen.

### Warum fällt es schwer, gute Software zu entwickeln?

Grundsätzlich sind oft bereits kleine Softwareprojekte schwer beherrschbar. In der Regel sind es jedoch insbesondere die großen bzw. komplexen, qualitativ hochwertigen Softwareprojekte, die eine Vielzahl von Teilnehmern, angefangen vom Kunden über ein Team von Software-Engineering-Spezialisten, Entwicklern, Testpersonal

und Wartungsspezialisten, benötigen. Jeder Projektteilnehmer bringt unterschiedliche Erfahrungen, Verfahren und Methoden mit, die miteinander abgeglichen und verwendet werden müssen. Unterschiedliche wirtschaftliche Faktoren genauso wie gesetzliche Aspekte müssen Berücksichtigung finden. In internationalen Projekten müssen beispielsweise oft Ländergrenzen, unterschiedliche Kulturen und Sprachen überbrückt werden.

Auch haben verschiedenartige Projektteilnehmer auch unterschiedliche Ziele. Kunden- und Benutzerwünsche liegen oft weit auseinander. Ein Projektleiter hat eine andere Sichtweise auf sein Projekt als ein Entwickler.



### Merke

Software-Engineering ist demzufolge eine Wissensquelle, von der alle Projektteilnehmer profitieren können, um ihre Ziele auch erreichen zu können.

Software-Engineering ist somit eine Art **Baukastensystem**, in dem **Standards, Verfahren** und **Methoden** angeboten werden, um einen möglichst erfolgreichen Projektverlauf zu gewährleisten. In dieses Baukastensystem sind das Wissen und die Projekterfahrungen aus zahlreichen, erfolgreichen und nicht-erfolgreichen Vorprojekten eingeflossen. Es bietet erfolgsorientierte Lösungsmöglichkeiten für den gesamten Ablauf eines Softwareprojektes an.

### Eigenschaften von Software

Welche Eigenschaften von Software erschweren die Erstellung von qualitativ hochwertiger Software?

Software ...

- ist **immateriell**
- wird nicht durch physikalische Gesetze begrenzt
- eine zugrunde liegende Modellbildung ist schwierig
- Anforderungen sind oft unzureichend geklärt, dokumentiert oder spezifiziert
- kann Defekte enthalten, Beispiele sind: Konstruktionsfehler, Spezifikationsfehler oder Portierungsfehler
- Ersatzteile gibt es nicht (nur evtl. sogenannte (engl.) „Patches“ (d. h. ein Austausch von Teilpassagen des Softwarecodes))
- ist schwer zu vermessen (Was ist gute und was ist schlechte Software?)
- ist leicht änderbar
- hat keinen Verschleiß, aber Software-„Aging“ (auf Deutsch: Alterung)
- Veränderungen sind fortlaufend nötig, um Software lauffähig zu halten.

### **Schwierigkeiten bei der Software-Erstellung**

Schwierigkeiten bei der Software-Entwicklung gibt es aufgrund deren Komplexität viele! Deshalb folgt hier keine vollständige Liste der zu bewältigenden Problematiken, sondern ein exemplarischer Ausschnitt möglicher Komplikationen:

- **Kommunikationsprobleme mit Projektbeteiligten:**  
Wenig Wissen über die Anwendung bei Software-Engineering-Spezialisten und Entwicklern; Anwender hat unklare Vorstellungen des Systems
- **Arbeitsabläufe werden durch Software oft verändert:**  
Akzeptanz- und Integrationsprobleme
- **Software-Varianten:**  
Konfiguration und Versionierung gestaltet Software viel komplexer
- **Software Einsatz in verschiedenen Umgebungen:**  
Portabilitätsprobleme.

Was kann man nun tun, um die genannten Probleme zu minimieren?

Die Antwort, die Software-Engineering-Spezialisten geben, lautet in etwa wie folgt:

Abhilfe schafft die Verwendung von:

- **Standards**  
Beispiel: Planen der Software durch Modellbildung
- **Methoden**  
Beispiel: Einsatz von Projektmanagement
- **Werkzeuge**  
Beispiel: Verwendung von UML-Tools, welche die Zeichenarbeit bei der Erstellung der Softwarearchitektur vereinfachen.

Genau diese Sammlung und Beschreibung der Standards, Methoden und Werkzeuge stehen im Fokus von Software-Engineering und damit auch im Mittelpunkt dieser Arbeit. Aus diesem Grund enthalten die weiteren Kapitel die kompakte Übersicht über die gebräuchlichsten und praxisrelevanten Verfahren dieses Fachgebiets.

## **■ 1.3 Geschichtlicher Überblick und die Folgen der Software-Krise**

In den fünfziger Jahren entstanden die ersten benutzbaren, jedoch teuren Computer, die Makros und bald Prozeduren niederer Programmiersprachen verstanden



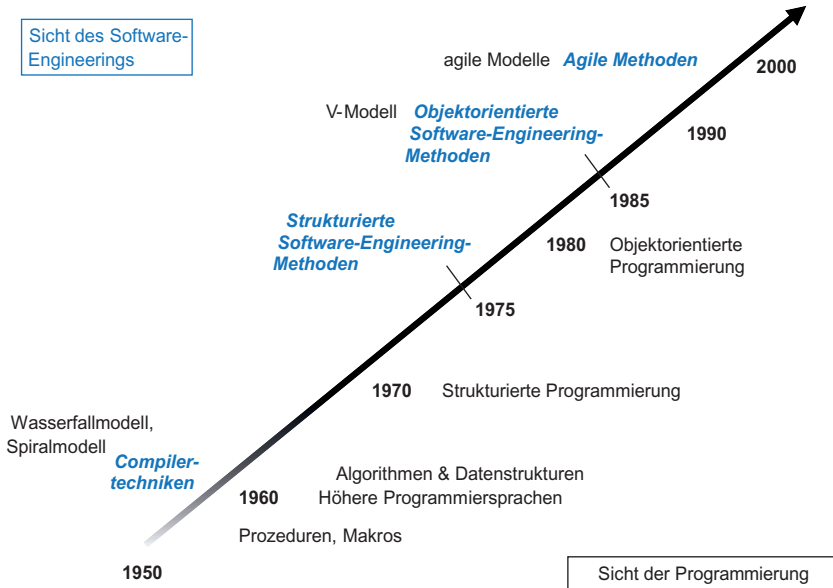


Abbildung 1.3 Software-Engineering-Zeitlinie

(siehe Abbildung 1.3). Diese funktionierten zunächst mit Lochkartentechnologie. Später folgten in den sechziger Jahren höhere Programmiersprachen, die mit ersten Compilern übersetzt wurden. In dieser Zeit überholten erstmals die **Kosten für Software**, durch gestiegene Komplexität und steigenden Umfang, die **Kosten für die Hardware**. Erste Software-Engineering-Methoden kamen auf. In dieser Zeit entstanden beispielsweise Vorgehensmodelle wie das Wasserfallmodell (publiziert von Royce 1970) (siehe Details in Kapitel 2.1.1). Algorithmen und Datenstrukturen wurden stetig komplexer und die strukturierte prozedurale Programmierung wurde zum Standard. Dies erforderte auch strukturierte Software-Engineering-Methoden wie beispielsweise das V-Modell (beschrieben in Kapitel 2.1.3) oder das Spiralmodell (publiziert von Böhm, 1988, IEEE) (siehe Kapitel 2.1.4). Als in den achtziger Jahren die objektorientierte Programmierung populär wurde, schlug sich das im Software-Engineering durch die Erfindung von grafischen Notationen, wie der UML (engl. Unified Modelling Language) zur Modellierung der Software, nieder.

Moderne und heute vielseitig eingesetzte Software-Engineering-Standards sind agile Methoden wie „Scrum“, „Extreme Programming“, agil-unterstützende Varianten wie „V-Modell XT“ und ähnliche (siehe Kapitel 2.1.5 – Agiles Modell). Agil heißt hier, dass Anforderungen für eine Software heutzutage jederzeit Änderungen unterworfen sein

können und das auf geänderte Kundenwünsche sehr viel schneller reagiert wird als mit anderen früher üblichen Vorgehensweisen. Mit den wichtigsten Anforderungen wird begonnen und die Lagerhaltung von Anforderungen wird möglichst minimiert. Dadurch können Anforderungen leichter später spezifiziert oder verändert werden.

### Die Software-Krise

Als ein Phänomen der Zeit entstand in den sechziger Jahren die sogenannte „Software-Krise“. Erstmalig erkannte man hier, dass die Kosten für die Software die Kosten für die Hardware übersteigen und dass dieser Trend durch steigende Komplexität und Umfang der Software anhalten wird. Erste große Softwareprojekte scheiterten und verursachten große monetäre Schäden.

Der Grund dafür war, dass die bisher genutzten undefinierten, undokumentierten Software-Engineering-Techniken nicht mit dem Umfang und der Komplexität der Software mithalten konnten.



#### Merke

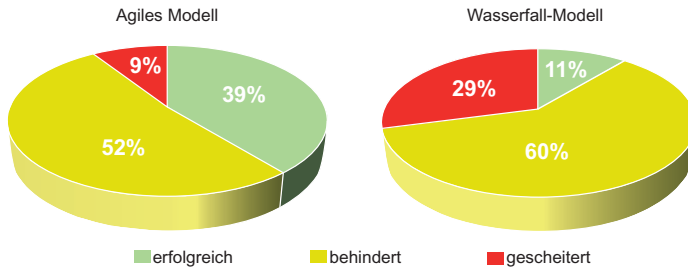
Auf einer NATO-Tagung (1968) wurde das Phänomen „Software-Krise“ diskutiert und als Reaktion der Begriff des **Software-Engineering** eingeführt.

Beispiele für sicherheitskritische Softwareprojekte mit Problemen gibt es viele; gestern wie heute. Genannt seien hier drei bekannte Vertreter zur Verdeutlichung der Relevanz des Einsatzes von Software-Engineering:

- Verstrahlung von Patienten durch Therac 25 (1985–87) u. a. durch fehlerhafte Prozesssynchronisation bei Messwerterfassungen
- Explosion der Ariane 5-Rakete (1996) u. a. durch den Speicherüberlauf eines Zählers
- Scheitern der Mars-Spirit-Mission (1999) durch einen Einheitenfehler im Navigationssystem.

Eine bekannte Publikation, die gerne zur Untermauerung der Notwendigkeit für Software-Engineering herangezogen wird, ist die Standish Group CHAOS-Studie (siehe Abbildung 1.4), bei der in der letzten Version über 10.000 IT-Projekte im Zeitraum zwischen 2011–2015 betrachtet wurden [SG15]).

Es zeigte sich, dass die Komplexität von Software-Projekten oft schwierig zu meistern ist. Die Gründe des Scheiterns von Softwareprojekten sind vielfältig. So geline



**Abbildung 1.4** Standish Group CHAOS-Studie 2011–2015 (>10.000 Software-Projekte jeder Größe)

gen auch heute im Mittel nur 11–39 % der IT-Projekte. Schon 52–60 % werden zumindest als „herausfordernd“ beschrieben und 9–29 % gelten gar als gescheitert. Dabei schneiden Projekte mit zugrunde liegender agiler Vorgehensweise (siehe Kap. 2.1.5) etwas besser ab als Projekte, die dem Wasserfallmodell (siehe Kap. 2.1.1) unterliegen. Dies ist gewiss darauf zurückzuführen, dass Änderungsanforderungen bei der agilen Vorgehensweise schnell und einfach integrierbar sind. Agiles Vorgehen wird daher derzeit als das modernste Vorgehensmodell angesehen.



### Merke

Weniger als die Hälfte aller Software-Projekte wird, durch Studien belegt, als „erfolgreich“ charakterisiert.

**Dies rechtfertigt in hohem Maße den Einsatz von Software-Engineering!**

Aus diesem Grunde werden in der Fachwelt folgende **Schlussfolgerungen** aus der Software-Krise gezogen:

- Früher war Software-Entwicklung ähnlich wie der Bau von Häusern **ohne** Architekten, Pläne und Maschinen.
- Software-Entwicklung ist keine kreative Kunst.  
(Die essenziellen **Ideen** für **gute Produkte** und deren **Anforderungen** zu haben ist jedoch sehr wohl  **kreativ!**).
- Software-Entwicklung ist demnach hauptsächlich eine **ingenieurmäßige Wissenschaft mit wohldefinierter Vorgehensweise**.